Popular Computing

VOL 3 NO 5
MAY 1975

006	370	888	386	557	418	945	257	873	927	166	340	205
772	885	145	496	453	897	857	644	949	776	964	318	974
276	362	496	039	608	543	749	774	683	390	544	779	675
281	302	505	018	890	065	698	191	834	016	368	224	242
424	245	363	876	057	209	041	059	789	975	487	315	880
101	281	713	631	151	591	801	524	027	359	515	980	075
970	303	787	595	439	605	672	082	231	332	623	531	439
418	476	166	401	562	430	936	515	609	280	985	307	503
029	029	380	275	502	389	001	969	727	291	423	797	265
633	009	540	159	602	880	054	480	992	376	173	531	893
085	591	399	724	874	466	279	013	377	135	511	709	922
719	224	072	378	326	535	320	395	504	769	064	170	971
416	308	558	710	862	114	222	512	447	129	911	377	915
455	188	072	558	196	613	620	269	283	354	466	777	332
425	617	302	034	137	561	408	847	864	954	177	052	350
734	911	578	200	373	695	370	786	737	864	445	285	783
684	473	377	364	427	257	344	834	893	575	649	412	676
250	405	976	909	226	081	400	008	596	811	624	229	365
535	897	932	384	626	433	832	795	028	841	971	693	993

Red Line — Blue Line

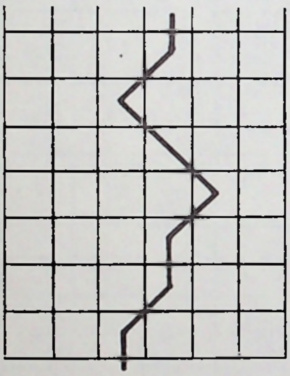
Red Line—Blue Line—A CONTEST

The cover pattern has 13 columns and 19 rows of squares, each containing a 3-digit number. A path is to be threaded from the red line to the blue line, proceeding from a square in one row to any of the three squares above it in the next row. Specifically, from the cell containing 028 in the bottom row, for example, the path may proceed to one of the cells containing 008, 596, or 811. At the edges, of course, there are only two choices instead of three. Thus, the path will proceed in the manner shown in the sample drawing.

The path may begin with any one of the 13 squares at the red line, and end with any one of the 13 squares at the blue line. The objective is to find the path for which the sum of the cell contents is the greatest. There are over 400 billion possible paths.

PROBLEM 91

For the person who finds a path with the greatest sum, POPULAR COMPUTING will pay $25 cash, or a two year subscription. Only one prize will be awarded. Entries must be received by July 31, 1975.



POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $18 per year, or $15 if remittance accompanies the order. For Canada and Mexico, add $4 per year to the above rates. For all other countries, add $6 per year to the above rates. Back issues $2 each.

The first article on Program Testing appeared in issue No. 11, February, 1974. That article dealt with the concept of program validation; we will now deal with the practice.

The very notion of validating the logic of a debugged program is probably the single most difficult idea to teach in all of computing. Beginners balk at learning it. (To be sure, beginners balk at lots of things. In their early stages, for example, they would rather write straight line code than learn to write loops.) But the opposition to program testing runs deeper. There is something--I know not what--in most people's minds that rejects the whole idea, and this even after several disasters occur that are directly connected to failure to test.

At the same time, the concept of validating a debugged program may be the most important single concept to be taught. Of what value is it to train expert coders in all the tricks and techniques of looping, subroutining, organizing, and documenting, if what their codes produce is garbage? In the worst case, beginners don't even know when they have produced garbage. In the next worst case, they suspect it, but can't prove it, and lack any method for tracking down the logical errors. In much the same way as an astronomer does not care how inaccurate a clock is, as long as he knows its error rate, so the expert computer programmer need not care (in the debugging and testing phases) if his program is in error as long as he knows that there are errors, and he knows how to find and correct them. Yes, it would be nicer to write error-free programs to begin with, but most of us cannot do that (those few who can are the masters) and must get by with the ability to produce an end result that is error-free. This is what leads to the art of testing.

The concept of program testing should be started early in a person's training and emphasized continuously. All during the design phases of a program, the question should be constantly in mind, "How am I going to test this program?" Indeed, quite frequently this question influences or even controls the design of a program.

The following statements are assumed to be true:

1. No program should be put into production without having been carefully tested.

2. The object of testing is to build up someone's confidence--usually that of the programmer who wrote the program--that the program does the task it is supposed to do, and that it will continue to do so as the data changes.

3. The test procedure must apply to the program being tested, and not to some other program that happens to be easier to test.

⇨

4. If any portion of the program's logic can be deleted and the test procedure still works, then the test is inadequate.

5. A proper test procedure requires positive feedback from the computer; that is, the running of the program with suitable test data should yield results that have been predicted; these results should be non-zero and should include values that could not readily occur by accident.

6. Miracles do happen. No test procedure is 100% perfect. The object is to validate the logic of the program and try to anticipate the errors that we all make every day.

7. Testing is mainly art; it is learned by practice.

8. All-alike data is usually weak for testing purposes.

9. The human eye cannot easily read and check many numbers. Sometimes we must do so, but usually the number of numbers or characters to be checked by eye should be kept small.

Just as with most new tools, program testing should be learned by applying it to simple problem situations first. Later, when large, complex programs are to be tested, they can be tested first at the subroutine level (and, by definition, you are then dealing with tiny problems), and then system tested in larger chunks. In the case of small, classroom problems, the test procedure requires more (and more complicated) code than the program being tested. The important thing is to get the logic of the test procedure straight.

Consider some of the consequences of <u>not</u> testing; they are all around us:

> "The program worked all right up to 37529 and then it blew up."
>
> "I assumed that the system's random number generator did what it was supposed to. How was I to know that every number it produced was 3 more than a multiple of 17?"
>
> "No one could have more than 12 dependents, so I never tested that part of the program."

--and so on, and on. Since most humans err on the side of testing too little, the list of distressing things that happen is long, and the list of pitiful excuses remains distressing.

Testing should be learned in the first introductory course in computing, and the point at which it should be introduced is when the student first sees a repetitive task. This is probably the point at which looping is brought up. (There is little point in devising an elaborate test for a straight-line Fortran program.) Suppose that the first exposure the beginner has to this concept is the task of adding the contents of 100 words of storage, or adding 100 floating point numbers in an array. Given what seems to be a logical analysis (e.g., a flowchart) and a debugged program (i.e., one that assembles or compiles, and executes), we are then ready to devise a test procedure.

The logic is as follows. The program should sum any 100 numbers in the data block. Therefore, it should sum 100 specific numbers. Therefore, we can control the data as we please, so that a sum can be reached that we can predict (say, by formula) which the program will calculate the hard way. An obvious solution is to load the data block with the numbers from 1 to 100 (and this data is readily generated). The expected sum can be predicted, by the formula for the sum of an arithmetic progression, as 5050. The complete test procedure is then the following:

1. Generate the consecutive numbers from 1 to 100 in the data block.
2. Run the program being tested.
3. Print the sum, and check it to be 5050.

If all of this works as stated, is the program now completely tested? Of course not. We have not, for example, tested for the ability to handle a zero (which is a troublesome number in many instances), or negative numbers, or overflow. One might consider altering the test data to run from -5 to +94 (with an expected sum then of 4450).

A number of small problem situations is given below. For each of them, a test procedure is to be constructed. Suggested procedures are given later. For some of the situations, a bad test procedure (labelled BTP) is given with the problem, and you might consider just why it's bad.

1. On a machine with a 32-bit word size, the number of 1 bits in a block of 100 consecutive words of storage are to be counted.

BTP: Load 50 words of the block with AAAAAAAA (hex) and the other 50 with 55555555 (hex). Run the program and print the count, which should be 1600.

2. On a machine with a 32-bit word size, a block of 100 words of storage contains positive integers. The sum of the integer square roots of those numbers is to be calculated.

BTP: Load the block with the consecutive numbers from 1 to 100. Run the program and print the sum, which should be 625.

3. Each card of a deck contains three 2-digit positive integers, representing the lengths of connected line segments. The line segments could form an equilateral triangle, an isosceles triangle, a scalene triangle, or no triangle. A program is to read a card, print its three values, and a statement about the type of triangle.

BTP: For the following input data, we expect the indicated results:

Read in	Print
35 35 35	Equilateral triangle
35 35 20	Isosceles triangle
30 40 50	Scalene triangle
80 20 20	No triangle

4. Each card of a deck bears two numbers in the form 060573,050676 (representing June 5, 1973 and May 6, 1976). A program is to read a card, print the two dates and the elapsed time in days between them. All dates are in the 20th century.

5. A program is written to accept an input value from a card, in the range from 00001 to 50000 (representing $0.01 to $500.00), output that number and its equivalent in words, as is done on checks. For example:

11759 **ONE HUNDRED SEVENTEEN AND 59/100 DOLLARS

BTP: Generate all input numbers from 00001 to 50000; run them through the program; run the output through the opposite conversion, from words to numbers; sum the resulting numbers; print that sum, which should be 1250025000.

6. The numbers in a block of 100 words of storage are to be reversed in order.

BTP: Generate the numbers from 1 to 100 in the data block. Run the program twice. Sequence check the block, to insure that each number is less than the one in the next higher addressed word.

7. The numbers in two blocks of storage, each 1000 words long, are to be interchanged; that is, the numbers in block A are to be moved in order to block B and the numbers in block B are to be moved in order to block A.

BTP: Generate the numbers from 1 to 1000 in block A and the numbers from 1000 to 1 in block B. Run the program. Sum the numbers now in both blocks, and check that both sums are 500500.

8. A subroutine is written to calculate arccosines, given a value for a cosine as an argument. For example, given .70710678 as an argument (half the square root of 2), the subroutine should return the value .78539819 (pi/4).

9. A block of 200 words of storage is addressed at K, K+1, K+2,...,K+199. The numbers at those addresses are to be moved in order to addresses K+1, K+2,...,K+200 (a pushdown list).

10. A triangular array of numbers is stored in consecutive words from T+1 through T+Y. The number of rows in the array is N, and Y is thus N(N+1)/2. The numbers are to be moved in storage so that the array is rotated 120° clockwise. N may have any value from 1 to 30. (Taken from the Penny Flipping Problem in PC23.) How shall this program be tested?

11. Recall the 3X+1 problem. For any positive integer, N, let X = N and follow this procedure. Replace X by X/2 if X is even. Replace X by 3X+1 if X is odd. Stop when X becomes one and count the number of terms generated, including the original term. (For example, for N = 9, the number of terms generated is 20.) A program was written to perform this algorithm on all values of N from 1 to 100000 and print a distribution of the resulting counts.

12. A program is written to calculate the first 10000 sets of Pythagorean Triplets; that is, sets of integers that satisfy the relation $X^2 + Y^2 = Z^2$. The program will generate values of R and T that meet all three of the following conditions:

a. R greater than T.
b. R and T having a greatest common divisor of one.
c. R and T of opposite parity; that is, one of them odd, the other even.

theory guarantees us that for such numbers

$$X = R^2 - T^2$$

$$Y = 2RT$$

$$Z = R^2 + T^2$$

will satisfy the Pythagorean relation. For example, for R = 12 and T = 7, the formulas give X, Y, and Z values of 95, 168, and 193. For a given value of R, the values of T will be taken in ascending order.

13. Records are to be read from two tapes, A and B, recorded in ascending order on some key in the record. Each tape has an end-of-file mark at the logical end of the file on that tape. The records are to be merged and written on tape C. The merge program is written to by-pass out-of-sequence records, indicate duplicate records, and check for missing end-of-file marks on the input tapes. Run controls in the program include record counts, hash totals, and a sequence check of the output stream. Outline a procedure for testing the merge program.

BTP: Write 500 records on tape A, with identifying keys of 2, 4, 6,...,1000. Write 500 records on tape B, with identifying keys of 1, 3, 5,...,999. Run the merge program. Sum the identifiers on tape C and check that the sum is 500500.

14. In 1957, the author and Charles F. Baker ran a calculation to sift out the first six million prime numbers, using what we thought was an ingenious algorithm. The calculation took over 100 hours of continuous use of an IBM 704. Due to a strange time limitation (the availability of the machine on a 4-day weekend) the program went into production with very little testing. What test procedures could have been used (and should have been used) had there been more time? (The final table, thoroughly validated, is available as <u>The First Six Million Prime Numbers</u>, The Microcard Corporation.)

15. In PC23-14 there appeared the COMMON DATA problem: Four blocks of storage each contain 1000 numbers. Write a program to count how many of the numbers (1) appear in all four blocks; (2) appear in any three of the blocks; (3) appear in any two of the blocks; and (4) appear in only one block.

16. Write a program to find all 10-digit numbers having the property that the digit in each position shows the number of occurrences of that digit in the number, according to the scheme:

$$N = \frac{x}{0}\ \frac{x}{1}\ \frac{x}{2}\ \frac{x}{3}\ \frac{x}{4}\ \frac{x}{5}\ \frac{x}{6}\ \frac{x}{7}\ \frac{x}{8}\ \frac{x}{9}$$

that is, the digit in the high order position is the number of zeros in N; the digit in the low order position is the number of 9s in N, and so on.

17. The Sets-of-Four problem was given in PC9-7, together with a flowchart of a possible solution. The problem is to examine successive sets of 4 words in storage (given 240 such sets, or 960 words) to insure that four types of numbers (call them A, B, C, and D) are present and in the order A, B, C, D. What data should be placed in the 960 words to insure that the program does its intended task?

18. Programs are written to solve the Penny Flipping problems I, II, and IV, given in issues 23 and 25. Devise test procedures for each of these three programs.

SUGGESTED TEST PROCEDURES AND COMMENTS

1. The problem is to find 100 numbers, all different, for which the number of 1-bits can be predicted without tedious human counting. One possibility is the integers from 1 to 100, for which a formula is available (PC11-11), giving a test count of 319. This test operates on only the low-order 7 bits of each word. A better test, then, would be the integers from 1 to 98, plus the number having all 1-bits, plus the number zero, for a total of 344.

2. The suggested procedure is logically correct; the sum of the integer square roots of the integers from 1 to 100 is 625. But why make things so difficult? Load the block with the squares of the integers from 1 to 100; run the program; check the resulting sum for 5050. This procedure will work also if the program operates on floating point numbers. The expected sum should then be close to 5050, allowing for rounding errors in the square root routine.

3. Neglecting the need for the program to edit the input data (e.g., check for negative values), the following is a minimum set of input values to test the logic adequately:

Input data A	Input data B	Input data C	Expected output
00	00	00	No triangle
50	50	50	Equilateral triangle
60	60	50	Isosceles triangle
60	50	60	Isosceles triangle
50	60	60	Isosceles triangle
30	40	50	Scalene triangle
50	30	40	Scalene triangle
40	50	30	Scalene triangle
10	80	20	No triangle
80	10	20	No triangle
10	20	80	No triangle
30	70	40	No triangle

4. For the elapsed days program, the following data and expected results is again a minimum set to test the program's logic:

Input data	Results	Comments
030375 030375	0	Same day
030375 030475	1	Minimum case
030475 030375	1	Minimum case, reverse order
030475 030476	366	Simplest leap year test
010100 123199	36523	Extreme case
121774 121784	3653	Readily calculated case
121784 121774	3653	Reverse case

5. The suggested procedure for the check writing program fails to test the essential part of the given program; namely, its ability to translate numbers into words. A representative sample of a dozen or so numbers that cover the range (and utilize every English word that could appear on the checks) should be run.

6. The fourth axiom says that if any portion of the logic of the program being tested can be eliminated, and the test still works, then it's a bad test. In this case, the entire program could be deleted.

The second half of the given procedure violates the fifth axiom. If it worked perfectly, there would still be no positive feedback from the machine.

Try this procedure: Generate the numbers from 1 to 100 in the data block. Run the program. Sum the contents of the data block, expecting the result 5050. Then print the contents of the first, fiftieth, and hundredth words in the block, for expected results of 100, 51, and 1, respectively.

7. Again, the given test will work with or without the given program, and hence tests nothing. A better procedure might be this: Generate the numbers from 1 to 1000 in block A, and the numbers from 1001 to 2000 in block B. Run the program, and sum the contents of both blocks. The sum for block A should then be 1500500, and for block B, 500500. Print the contents of the first and last words of both blocks, to check for 1001, 2000, 1, and 1000 respectively.

8. Generate the values from -1.1 to +1.1 at intervals of .1 and feed these 23 values to the subroutine. Calculate the cosine of the output from the subroutine, using the built-in cosine function. The results, printed out, should show the following:

Input value	Output result
-1.1	Invalid argument
-1.0	-1.0
-0.9	-0.9
-0.8	-0.8
.	.
.	.
0.0	0.0
0.1	0.1
.	.
.	.
1.0	1.0
1.1	Invalid argument

(The results in the right hand column will not, in general, be precise, but the error will give an indication of the precision level of the subroutine.) In addition, feed the subroutine the values .70710678 and .86602540 and check for calculated results of .78539816 and .52359878.

9. Generate the numbers from 1 to 200 in the data block from K through K+199. Run the program. Sum the contents of words K+1 through K+200, for an expected result of 20100. Print the contents of words K, K+1, K+199, and K+200, for expected results of 1, 1, 199, and 200 respectively.

10. Set N = 10; that is, fix the array size at 10 rows, or 55 elements. Generate the numbers from 1 to 55 in words T+1 through T+55. Run the program. Sum the values now in T+22 through T+28 (this is the 7th row of the array) for an expected result of 189. Sum the values in T+10, T+14, T+19, T+25, T+32, T+40, T+49 (this is the 4th column of the array) for an expected result of 175. Print the contents of words T+1, T+46, T+55, which should be 46, 55, and 1, respectively.

11. The 3X+1 distribution. Now we are getting into more realistic situations, where things can't be controlled so neatly. Limited runs should be made with the program, in ranges where results are already known (and this process chains back eventually to some hand calculation). The ability of the program to handle overflow situations for values of X that are generated along the way, and to insure that individual counts do not exceed the limits set (the largest known count is 706) must be carefully checked, perhaps by forcing the program to go wrong at those points. Beyond such things, such a program is proceeding into the unknown.

12. Here again, a program is proposed that will probe the unknown. Suppose that the longest table of Pythagorean Triplets previously calculated ran to 1000 lines. Then the 1000th line produced by <u>this</u> program should agree (and show X = 1419; Y = 8260; and Z = 8381). The heart of the program is the logic of generating the R and T values properly. For R = 60, for example, the T values should be 1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, and 59.

Notice that it is useless to check that the program's normal output conforms to the Pythagorean relation, since the formulas used guarantee that that will be true, even if the R and T values do not meet the required conditions. For example, R = 9 and T = 15 are values that violate all three conditions, but the formulas produce X, Y, and Z values of -144, 270, and 306, which will satisfy the Pythagorean relation.

13. The suggested test for the merge program merely checks that all the input numbers have moved to the output tape. Even if it is also determined that the output records are in sequence, it has not been determined that a <u>merge</u> took place; the program could have performed an <u>interfiling</u> operation, which is a much simpler task. The following data could provide a better test:

A	B
1	2
3	4
5	10
7	12
11	14
9	20
13	22
19	18
EOF	EOF

The sum of the numbers on the output tape should be 143; records 9 and 18 should be by-passed as out of sequence; the output records should be in sequence; record counts should show 8 for A, 8 for B, 14 for C, and 2 by-passed.

14. Preliminary runs with the prime sieve were spot checked against existing tables of primes before the run of 6000000 was made. Even with this, there would still remain nagging doubts as to the accuracy of the results:

(a) Some primes could have been skipped over, due to things like incorrect loop tests in the high ranges.

(b) Some composite numbers could have been certified as primes in the same way.

(c) Even if the program's logic was correct, there could have been intermittent machine failure which could cause a correct number to be outputted incorrectly.

The output of the long run seemed in good order (all spot checks checked; 5999998 of the output numbers ended in 1, 3, 7, or 9; no difference between successive primes exceeded 220; the distribution of the primes among the natural numbers appeared smooth, and so on) but the first reassurance we had came from Prof. D. H. Lehmer. We had told Prof. Lehmer of our adventures and sent him our 6000000th prime (104395289) and he wrote back "That's right." He had programmed Meissel's formula, which gives essentially the position number of any given prime, fed that program the number 104395289, and received back the result 6000000. This was comforting, since it ruled out the possibility of systematic errors in logic that would have invalidated the table.

Prof. Lehmer also hinted gently that we had wasted a great deal of machine time by pointing out a more efficient algorithm (see PC13-7). In 1959, using his better algorithm, a run was made on an IBM 7090 to recalculate all 6000000 primes and compare against the 1957 results. There were no discrepancies, and the 7090 run took 21 minutes. The machine, the algorithm, and the programmer were all different for the check run. That makes a pretty good test procedure, albeit ex post facto.

15. This procedure might be adequate: Clear all 4000 words to zero. Load the following numbers into the specified blocks (anywhere in the blocks):

Block A:	9999	9998	9997	9996	9994	9993
	9991	9988	9987	9987	9987	9987
Block B:	9999	9998	9997	9995	9994	9992
	9990	9988				
Block C:	9999	9998	9996	9995	9993	9992
	9989	9988				
Block D:	9999	9997	9996	9995	9991	9990
	9989	9988				

Then load into every word still containing zero the consecutive numbers from 1 to 3964. Run the program and check for these expected results:

Number of numbers appearing in all 4 blocks:	2
Number of numbers appearing in 3 blocks:	4
Number of numbers appearing in 2 blocks:	6
Number of numbers appearing in 1 block:	3968

16. A test procedure for this program? There shouldn't be a program at all; the unique number being sought can be reasoned in a few minutes: 6210001000.

17. The test data for the Sets-of-Four problem should include all permutations of correct sets, and those sets should be printed out after running the program to insure that the sorting has taken place; plus sets of data for the forms ABCE, ABEC, AEBC, EABC (where E stands for some other type) as well as sets of the forms AABC, BABC, ACCC, DDDD. Each test run should have a full 240 sets of data, and the set numbers of the incorrect sets should be predicted before running the test.

18. Adequate test data for the various Penny Flipping problems has been presented from previous calculations--and those results were validated by extensive hand calculations. □

26

Log 26	1.414973347970817964420244052666821457597919069849177
Ln 26	3.258096538021482045470719563023495172880768079120462
$\sqrt{26}$	5.099019513592784830028224109022781989563770946099596
$\sqrt[3]{26}$	2.962496068407370508673062189341838537566357422318866
$\sqrt[5]{26}$	1.918645191625306247864278567185733088421506528694289
$\sqrt[10]{26}$	1.385151685421241599502813326193240889758722178108777
$\sqrt[100]{26}$	1.033117536511968663502026341041637539220370693437179
e^{26}	195729609428.8387642697763978760953427920361009506976
π^{26}	8431341691876.207066643299322156645686147915492941130

N-SERIES

Book Review

(1) Programming Proverbs

(2) Programming Proverbs for Fortran Programmers

by Henry F. Ledgard, Hayden Books, paper covers, 130 pages each, $5.65 each.

These two are the same book, promoting the essentials of good programming (in the sense of coding) practices. The first named book uses Algol and PL/I as the reference languages.

This, then, is the fifth book on programming style. Three others were reviewed in issue No. 13:

Fortran Techniques, A. Colin Day, 1972.
The Elements of Fortran Style, Kreitzberg and Schneiderman, 1972.
The Elements of Programming Style, Kernighan and Plauger, 1974.

and the fourth book is:

Program Style, Design, Efficiency, Debugging, and Testing, Dennie Van Tassell, 1974.

The Kreitzberg and Schneiderman book, the Kernighan and Plauger book, and Ledgard's books all pay tribute to W. S. Strunk's The Elements of Style.

Time will tell which of these books will prove to be the Strunk of computing (and/or which will become popular with programmers or students). At the moment, the book by Kernighan and Plauger leads the parade.

Ledgard condenses much of the distilled wisdom that has accumulated in the form of 25 "proverbs." Some of them are axiomatic and bear endless repeating to novices:

AVOID UNNECESSARY GOTO'S.
DO NOT RECOMPUTE CONSTANTS WITHIN A LOOP.
AVOID TRICKS.

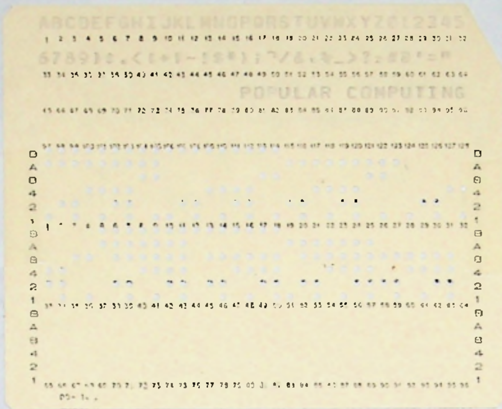
Others, like

REREAD THE MANUAL.
CONSIDER ANOTHER LANGUAGE.

while sound, are of doubtful value. The pair

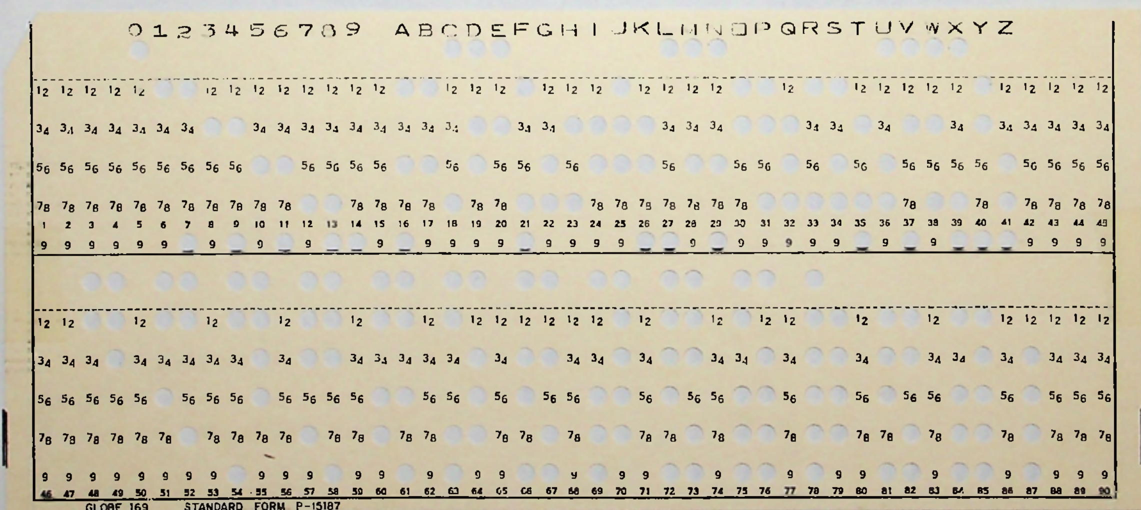
Text continued on page 16

The Other Punched Cards



The 80-column punched card must be familiar to nearly everyone--in or out of computing. The 96-column card above is the primary input medium for the IBM System/3. The hole combinations for the digits and letters form a systematic pattern.

The 90-column card of Remington-Rand, below, is not so familiar. It traces from the original Powers devices. The hole combinations for the digits and letters form an erratic pattern.



GET THE PROGRAM CORRECT BEFORE TRYING TO PRODUCE GOOD OUTPUT.
WHEN THE PROGRAM IS CORRECT, PRODUCE GOOD OUTPUT

are awkward and pithless; wiser souls long ago put it this way:

FIRST MAKE IT WORK, THEN MAKE IT PRETTY.

And the maxim:

PRETTYPRINT

is parochial and cutesy.

However, all 25 reflect basic truths and painful learning by those who preceeded us. Each maxim is expanded with text exposition (very sound and down to earth) plus examples showing the wrong way and the right way.

Ledgard's books (particularly the pink one on Fortran) could become quite popular with teachers of computing. The subtitle is Principles of Good Programming with Numerous Examples to Improve Programming Style and Proficiency. Anything that will foster those goals is to be commended. □

PROBLEM 92

Euler Spoilers

In 1769, the mathematician Euler conjectured that the equation

$$J^5 + K^5 + L^5 + M^5 = N^5$$

had <u>no solutions</u> in positive integers. This conjecture stood until 1966, when Lander and Parkin used a Control Data 6600 computer to find a solution. They found four solutions, for one of which the value of N is less than 75.

Since we know a solution (J, K, L, M, N) exists in positive integers, the largest of which is less than 75, you should be able to program a computer to rediscover this solution, and perhaps some of the others.

You may learn a bit about computing by tackling this problem even though a partial answer is already known. Euler also conjectured that the equation

$$I^4 + J^4 + K^4 = N^4$$

has no solution in positive integers. As far as I know, that conjecture still has not been proved or disproved. Do you wish to hunt for a counterexample?

--RICHARD V. ANDREE